Éléments d'architecture des ordinateurs : travaux pratiques d'assembleur sur PC Intelx86 sous Linux

Laboratoire d'Informatique fondamentale de Lille
Université des sciences et technologies de Lille
F-59655 Villeneuve d'Ascq CEDEX France

1er octobre 2007

Ces notes constituent un support de travaux pratiques pour un cours d'architecture élémentaire des ordinateurs. Lors de ces séances, il s'agit principalement de programmation assembleur sur des ordinateurs personnels d'architecture intel 32 bits gérés par le système d'exploitation Linux. L'assembleur utilisé est le gnu assembleur et nous utiliserons le gnu debugger pour l'exécution pas à pas et l'examen de la mémoire et des registres.

Ce texte — comme son titre l'indique — vise deux objectifs :

- fournir une synthèse « élémentaire » des notions requises par un débutant pour l'apprentissage de l'assembleur ;
- donner une représentation architectonique[1] du sujet.

Ainsi, ces notes ne prétendent ni à l'exhaustivité — toutes les instructions et possibilités de l'assembleur ne sont pas présentées — ni à apprendre à écrire du code optimal. Cet enseignement est plutôt orienté par la perspective d'en faire une base permettant de comprendre comment un programme — en C par exemple — est exécuté par la machine.

Un bref descriptif du jeu d'instruction des processeurs 32 bits de type intel est disponible en annexe C (pour plus d'information voir [2]). De plus, en utilisant la commande `info as` dans un interpréteur de commande, on accède au manuel d'utilisation de gas.

Conventions typographiques. Dans ce document, nous utilisons les conventions suivantes :

`foo` représente la commande foo dans un interpréteur de commandes ;
foo représente la commande foo dans un environement — par exemple gdb ;
`foo` représente une instruction assembleur.

Le guide d'utilisation de gas. Ce document n'est pas un manuel présentant le gnu assembleur ; pour ce faire, nous suggérons l'utilisation du guide d'utilisation accessible depuis un interpréteur de commandes par l'instruction `info gas`.

[1] qui à rapport à l'architecture

Chapitre 1

Prise en main de l'assembleur gas

L'objectif de cette section est d'écrire, compiler et exécuter — sans trop de douleur — son premier programme assembleur.

1.1 Descriptif sommaire de la famille des processeurs Intel iAPx86

Il s'agit d'une famille de processeurs à compatibilité ascendante i.e. le code des plus anciens est toujours compris et correctement exécuté par les plus récents. Dans cette famille, on peut citer :

1978 le 8086 dispose d'un bus de 16 bits pour les données. Sa capacité d'adressage est de 1 Mo et il est cadencé à 4.77 ou 8 Mhz ;

1979 le 8088 est la version 8 bits du 8086. il a les mêmes caractéristiques, mais un assembleur plus réduit ;

1982 le 80286 est une machine 16 bits, pouvant adresser 8 Mo de code et de données, cadencé à $6, 8, 10, 12$ ou 16 Mhz. Il introduit deux modes d'adressage (réel ou protégé) ;

1985 le 80386 est un faux 32 bits, adressant 4 Go et cadencé à $16, 20$ ou 25 Mhz ;

1989 le 80486 est un vrai 32 bits doté d'une mémoire cache intégrée et d'une unité de calcul en virgule flottante ;

1993 le 80586, appelé pentium pour des raisons de protection commerciale, dispose d'un bus de données de 64 bits et est muni d'un dispositif de prévision des branchements. Il est constitué de 2 processeurs en pipe-line parallèles lui permettant d'exécuter deux instructions en même temps. Initialement cadencé à 66 Mhz, son cadencement à 200 MHZ est mis sur le marché en 1996 ;

1995 le Pentium Pro est constitué de 6 millions de transistors (contre 275000 pour le 386, 1 million pour le 486 et 3 pour le pentium) et la dimension des traits de ses transistors est de 0.6 microns (contre 11.5 pour le 386). Il peut traiter 250 millions d'instructions par seconde ;

2000 le Pentium III est cadencé à 1 Ghz.

En 2001, l'Itanium est lancé ; c'est un processeur disposea d'une architecture 64 bits. Les processeur X86 disposent maintenant d'une extention 64 bits à l'architecture IA32, mais sans rapport avec l'IA64 de l'itanium.

Ces architectures ne seront pas abordé dans ce cours.

1.2 Registres généralistes de la famille des processeurs 32 bits Intel

Pour manipuler des données, les processeurs 32 bits de type Intel possèdent quatre registres de 32 bits à usage généraux nommés %eax, %ebx, %ecx et %edx. Il est possible de n'utiliser qu'une portion de ces registres constituée de 16 bits : dans le cas du registre %eax cette portion est nommée %ax (les processeurs intels était à l'origine d'une capacité de 16 bits ; le `e` de %eax signifiant extended).

Pour finir, notons que chacun de ces registres 16 bits peut être décomposer en deux registres de 8 bits. Par exemple, le registre %ax est constitué des registres %ah et %al ("High" et "Low" respectivement). Ce principe est illustré par la Figure 1.1.

registres 16 et 8 bits

bits	15 8	7 0
%ax	%ah	%al
%bx	%bh	%bl
%cx	%ch	%cl
%dx	%dh	%dl

registres 32 bits

bits	31 16	15 0
%eax		%ax
%ebx		%bx
%ecx		%cx
%edx		%dx

Fig. 1.1 – Les registres généralistes d'un processeur Intel 32 bits

Remarque 1. Ces registres ne sont pas tous indépendants : si le registre `AL` est modifié, il en est de même des registre `AX` et `EAX` alors que le registre `AH` n'est pas affecté (ainsi que les autres registres).

Remarque 2. Des registres 64 bits sont formés par la conjonction de 2 registres 32 bits (%edx :%eax) et utilisés lors de certaines opérations (`mul`).

1.3 Exemple d'instruction assembleur : le mouvement de données

Pour stocker des informations dans un registre, nous disposons de la famille d'instructions assembleur `mov`. Par exemple, pour stocker la valeur décimale 2 dans le registre `EAX` (resp. `BX`, `CH`), on écrit `movl $2,%eax` (resp. `movw $2,%bx`, `movb $2,%ch`).

Dans cet exemple, on utilise la syntaxe ATT dans laquelle :
- le préfixe $ indique une opérande ne nécessitant pas d'interprétation (ici un entier) ;
- les noms des registres sont préfixés par le symbole %. Ceci permet d'utiliser du code assembleur directement dans un programme C sans risque de confusion avec un nom de macro ;
- la première opérande de l'instruction `mov` est la source et la seconde est la destination (la syntaxe Intel utilise la convention inverse) ;
- la taille des opérandes se lit sur le suffixe de l'instruction :

b	pour byte	:	8 bits ;
w	pour word	:	16 bits ;
l	pour long	:	32 bits.

  Le compilateur n'impose pas strictement l'usage du suffixe i.e. l'instruction `mov $2,ch` est valide tant qu'il est possible de déduire la taille des données déplacées (dans le doute, on utilise l'architecture 32 bits par défaut).

Remarque 3. Il existe plusieurs variantes de l'instruction `MOV` dont voici quelques exemples :

```
mov $1515, %AX
mov %BL, %DH
```

Nous reviendrons sur ce sujet lors de l'étude du thème 4. Pour l'instant poursuivons la présentation des notions nécessaires à l'écriture de notre premier programme assembleur.

1.4 Structure d'un programme assembleur

Nous utiliserons dans cette section un fichier source assembleur d'un programme faisant la somme de 5 entiers stockés en mémoire et qui stocke le résultat en mémoire ; ce fichier nous permettra de nous familiariser avec la structure d'un programme assembleur.

Remarque 4. Tout texte compris entre les symboles `/*` et `*/` est un commentaire qui n'est pas pris en compte par le compilateur. Il existe bien d'autres délimiteurs de commentaires (; par exemple) dépendant de l'architecture et qui sont décrit dans le manuel de *gas*.

Le fichier source d'un programme assembleur est divisé en — au moins — deux segments :

1. Le segment de données.
2. Le segment de texte qui contient le code source assembleur.

1.4.1 Segment de données

Ce segment commence par la ligne

```
.data
```

Remarque 5. Tout nom commençant par `.` est une *directive* de compilation (une instruction au compilateur et non pas une instruction que le compilateur doit traduire en langage machine). L'ensemble des directives est disponible dans la section `Pseudo Ops` du manuel en ligne `info as`. Dans notre cas, il s'agit de préciser que ce qui suit sont des données.

Pour organiser les segments, on utilise des *labels* aussi appelé *étiquette*. Ainsi, pour stocker des données — 5 entiers longs par exemple et une chaîne de caractères — le fichier source assembleur doit contenir dans son segment de données le texte suivant :

```
UnNom :
      .long 43    /* d\'efinition d'un entier cod\'e sur~$4$ octets */
      .long 54
      .long 23    /* on aurait pu d\'efinir cette suite d'entiers  */
      .long 32    /* par .long 43,54,32,76                         */
      .long 76
      .string "hello world" /* Des donn\'ees dont on ne se sert pas */
      .float 3.14
```

Les directives `.long` indiquent le codage des entiers sur 32 bits et la directive `.string` indique qu'une suite d'octets consécutifs sera réservée et instanciée en mémoire lors de l'exécution (un octet pour le code ascii de chaque lettre de la chaîne plus un dernier octet initialisé à 0). Notez que pour coder un flottant, on utilise la directive `.float`.

Le label `UnNom` peut être utilisé dans le segment de code afin de référencer les données correspondantes (cf. section suivante).

Remarque 6. Il est possible de réserver de l'espace non initialement affecté ; par exemple le texte :

```
UnAutre:
          .space 4
```

permet de réserver 4 octets en mémoire.

1.4.2 Segment de texte

Ce segment commence par la directive

```
.text
```

Afin d'exécuter le code résultant de la compilation de notre programme assembleur, il faut définir un *point d'entrée* i.e. un point de départ. Pour ce faire, on utilise la directive `.globl` et un label. Par exemple, le code :

```
.globl _start
```

indique que le label `_start` peut être accessible par tous programmes l'utilisant à condition que *l'édition de liens* entre ce programme et le programme que nous sommes en train d'écrire aura été faîte.

Par défaut, l'exécution du programme commencera par l'instruction assembleur immédiatement postérieure au label `_start`. Ce dernier label est le point d'entrée mais vous pouvez en spécifier un autre (`main` par exemple) à condition d'utiliser l'option `-e` de l'éditeur de liens `ld` (voir section 1.5.1).

Dans notre exemple, le code source immédiatement postérieur au point d'entrée est :

```
_start:
        movl $5, %eax      /* EAX va nous servir de compteur indiquant le
                              nombre d'entiers restant \`a additionner   */
        movl $0, %ebx      /* EBX va contenir la somme de ces entiers    */
        movl $UnNom, %ecx  /* ECX va << pointer >> sur l'\'el\'ement
                              courant \`a additionner                   $*/
```

Cette dernière instruction place dans le registre %ecx l'*adresse* associée au label `UnNom`. La mémoire de données est vue comme une suite d'octets indexée de 0 à n :

offset	0	...	x	x+1	x+2	x+3	x+4	x+5	x+6	x+7	...	x+20	x+21	...
octet	?	...	43				54				...	104	101	...

Le label `UnNom` sert de référence au déplacement relatif — à l'offset — x définit par le système. Ainsi :

– la commande `movl $UnNom, %ecx` place l'offset x dans le registre %ecx ;
– la commande `movl UnNom, %ecx` place l'octet situé à l'offset x du segment de mémoire dans le registre %ecx ;
– on trouve à l'offset x+5 la valeur 104 qui n'est que le code ascii pour la lettre h. Notez qu'il n'y a pas de typage à ce niveau en assembleur.

Comme dans le cas du segment de données, plusieurs labels peuvent être utilisés dans le segment de code. Par exemple, on peut faire suivre le code source ci-dessus par les instructions :

```
top:   addl (%ecx), %ebx  /* Additionne au contenu de EBX ce qui est
                             point\'e par ECX et stocke le r\'esultat dans
                             EBX                                         */
        addl $4, %ecx      /* D\'eplace le << pointeur >> sur
                              l'\'el\'ement suivant                      */
        decl %eax          /* D\'ecr\'emente le compteur EAX             */
        jnz top            /* Si le contenu de EAX est non nul alors
                               ex\'ecuter le code \`a partir du label top */
done:   movl %ebx, UnAutre /* sinon, le resultat est stock\'e        $ */
```

L'instruction `addl (%ecx), %ebx` additionne le contenu du segment de données dont l'offset correspond au contenu du registre %ecx avec le contenu du registre %ebx ; le résultat est stocké dans le registre %ebx. L'instruction `decl %eax` décrémente ce registre ; si le contenu de ce registre est différent de 0, alors la commande `jnz top` provoque un saut dans le flux d'exécution des instructions et l'instruction associée au label `top` est exécuté. Sinon, c'est l'instruction suivante dans le flux d'instruction qui est exécutée. Un mécanisme de boucle conditionnelle est ainsi mis en place par ce code source (nous reviendrons sur ce point lors de l'étude du thème 2).

Remarque 7. Les instructions suivantes

```
        movl    $0,%ebx      /* Ces instructions permettent d'invoquer de */
        movl    $1,%eax      /* terminer l'ex\'ecution d'un programme     */
        int     $0x80        /* assembleur et sont indispensable         $*/
```

doivent impérativement terminer votre code source. Dans le cas contraire, le processeur continuera d'exécuter le segment de code bien après la fin de votre dernière instruction. Pour éviter ce comportement qui ne peut que se terminer par une faute de segmentation, il convient de faire appel au système d'exploitation pour terminer l'exécution de votre programme en utilisant le bout de code ci-dessus (nous reviendrons sur ce point lors de l'étude du thème 3).

Synthèse. Pour conclure, nous obtenons donc le code source suivant :

```
.data
UnNom :
      .long 43    /* d\'efinition d'un entier cod\'e sur~$4$ octets */
      .long 54
      .long 23    /* on aurait pu d\'efinir cette suite d'entiers  */
      .long 32    /* par .long 43,54,32,76                         */
      .long 76
      .string "hello world" /* Des donn\'ees dont on ne se sert pas */
      .float 3.14
UnAutre:
          .space 4
.text
.globl _start
_start:
         movl $5, %eax      /* EAX va nous servir de compteur indiquant le
                               nombre d'entiers restant \`a additionner   */
         movl $0, %ebx      /* EBX va contenir la somme de ces entiers    */
         movl $UnNom, %ecx  /* ECX va << pointer >> sur l'\'el\'ement
                               courant \`a additionner                    */
top:   addl (%ecx), %ebx  /* Additionne au contenu de EBX ce qui est
                             point\'e par ECX et stocke le r\'esultat dans
                             EBX                                          */
         addl $4, %ecx      /* D\'eplace le << pointeur >> sur
                               l'\'el\'ement suivant                      */
         decl %eax          /* D\'ecr\'emente le compteur EAX             */
         jnz top            /* Si le contenu de EAX est non nul alors
                              ex\'ecuter le code \`a partir du label top */
done:    movl %ebx, UnAutre /* sinon, le resultat est stock\'e
        movl    $0,%ebx     /* Ces instructions permettent d'invoquer de */
        movl    $1,%eax     /* terminer l'ex\'ecution d'un programme     */
        int     $0x80       /* assembleur et sont indispensable          */
/* Utilisateur d'emacs, veillez \`a ajouter une ligne vide apr\`es
ce commentaire afin de ne pas provoquer un Segmentation fault */
```

Ce code peut être saisi dans votre éditeur de texte préféré — emacs ou vi — afin de produire un fichier source. Par convention, on acolle à ce type de fichier source le suffixe `.s` : dans notre cas, nous allons appeler ce fichier `sum.s`.

1.5 Compilation et exécution pas à pas

Il nous reste à rendre ce fichier source — compréhensible par un être humain convenablement formé — compréhensible par un ordinateur grâce à la compilation.

1.5.1 Compilation

Dans notre contexte, la compilation se décompose en 3 phases successives :

1. **Le traitement par le préprocesseur :** le fichier source est analysé par un programme appelé *préprocesseur* qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source, etc.) ;
2. **L'assemblage :** cette opération transforme le code assembleur en un fichier *binaire*, c'est à dire en instructions directement compréhensibles par le processeur. Le fichier produit par l'assemblage est appelé fichier *objet* ;
3. **L'édition de liens :** un programme est souvent séparé en plusieurs fichiers source (ceci permet d'utiliser des librairies de fonctions standard déjà écrites comme les fonctions d'affichage par exemple). Une fois le code source assemblé, il faut donc *lier* entre eux les différents fichiers objets. L'édition de liens produit alors un fichier *exécutable.*

Pour assembler un fichier source assembleur — disons `sum.s` — vous pouvez utiliser la commande suivante dans votre interpréteur de commande :

```
as -a --gstabs -o sum.o sum.s
```

Nous venons d'utiliser un certains nombre d'options

`-o` indique que le nom du fichier objet — dans notre cas `sum.o` ;

`-a` permet d'afficher le code source que nous venons d'assembler parallèlement aux codes assemblés et aux offsets correspondant ;

`-gstabs` n'est utilisé que pour l'exécution pas à pas de notre exécutable.

Pour obtenir un fichier exécutable `sum`, il nous faut mener à bien l'édition de liens en utilisant la commande :

```
ld -o sum sum.o
```

Voila, nous pouvons exécuter ce programme en invoquant le nom de l'exécutable dans votre interpréteur de commandes favori... si ce n'est que le résultat du programme (stocker une somme dans le segment de données d'un processus) ne nous est pas visible. Pour voir ce qui se passe, nous allons utiliser un outil d'exécution pas à pas.

1.5.2 Exécution pas à pas dans l'environnement gnu debugger

Pour ce faire nous allons utiliser le gnu debugger `gdb` dans un premier temps puis l'interface graphique `ddd`[1]. Un récapitulatif des principales commandes de gdb est disponible dans l'annexe E.

[1] L'auteur n'arrivant pas à utiliser correctement cette interface, peu de place y sera consacrée dans ces notes.

L'environnement `gdb` permet d'exécuter des programmes pas à pas et d'examiner la mémoire. Il dispose d'un guide utilisateur accessible par la commande `info gdb`. Pour utiliser gdb, il suffit de l'invoquer dans son interpréteur de commandes en lui indiquant le fichier à examiner :

```
[espoir.lifl.fr-sedoglav-/home/.../Sources] gdb sum
GNU gdb 5.3-22mdk (Mandrake Linux)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-mandrake-linux-gnu"...
(gdb)
```

Ce programme propose une aide en ligne accessible par la commande help :

```
(gdb) help help
Print list of commands.
(gdb) help quit
Exit gdb.
```

Exécution du programme

Le programme considéré peut être exécuté dans l'environnement `gdb` en utilisant la commande `run`.

```
(gdb) run
Starting program: /home/.../Sources/sum
Program exited normally.
(gdb)
```

Comme lors de l'exécution par le biais d'un interpréteur de commande, le fonctionnement de notre programme ne nous est pas apparent. Pour ce faire, nous allons forcer le programme à stopper et contrôler son exécution.

Points d'arrêt

Lorsque le code source de l'exécutable est disponible la commande `list` permet d'afficher le code source avec chacune de ces lignes numérotées. Dans notre cas :

```
(gdb) list
1       .data
2
3       UnNom :
4               .long 43
```

La commande `break` permet de placer un point d'arrêt sur une instruction du programme source de manière à ce qu'à la prochaîne exécution du programme dans `gdb`, l'invite du dévermineur soit disponible avant l'exécution de cette instruction.

Une instruction du programme source peut être repérée par le numéro de ligne correspondant ou par un label. Ainsi, la suite de commande :

```
(gdb) break 26
Breakpoint 1 at 0x8048079: file sum.s, line 26.
(gdb) break top
Breakpoint 2 at 0x8048083: file sum.s, line 29.
```

permet de placer deux points d'arrêts aux endroits spécifiés. On peut avoir la liste des points d'arrêts en utilisant dans `gdb` la commande `info` :

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048079 sum.s:26
2   breakpoint     keep y   0x08048083 sum.s:29
```

Exécution pas à pas

Une fois ceci fait, on peut exécuter notre programme dans l'environnement `gdb` :

```
(gdb) run
Starting program: /home/.../Sources/sum

Breakpoint 1, _start () at sum.s:26
26              movl $0, %ebx      /* EBX va contenir la somme de ces entiers  $  */
Current language:  auto; currently asm
(gdb)
```

On constate que cette fois l'invite `(gdb)` se présente avant la fin normal du programme.

Pour provoquer uniquement l'exécution de l'instruction `movl $0,%ebx` , on peut utiliser la commande `step` :

```
(gdb) step
27              movl $UnNom, %ecx  /* ECX va << pointer >> sur l'\'el\'ement $
(gdb)
```

Pour provoquer l'exécution de l'ensemble des instructions comprises entre la position courante et le prochain point d'arrêt, on peut utiliser la commande `continue`. Ainsi, si on reprend le processus ci-dessus depuis le début :

```
[espoir.lifl.fr-sedoglav-/.../Sources] gdb -q sum
(gdb) break 26
Breakpoint 1 at 0x8048079: file sum.s, line 26.
(gdb) break top
Breakpoint 2 at 0x8048083: file sum.s, line 29.
(gdb) run
Starting program: /home/.../Sources/sum
c
Breakpoint 1, _start () at sum.s:26
26              movl $0, %ebx      /* EBX va contenir la somme de ces entiers     $  */
Current language:  auto; currently asm
```

```
(gdb) continue
Continuing.

Breakpoint 2, top () at sum.s:29
29      top:   addl (%ecx), %ebx
(gdb)
```

En combinant ces commandes, on peut pas à pas exécuter toutes les instructions du programme jusqu'à sa fin naturelle. Il ne nous reste plus qu'à jeter un coup d'oeil dans les entrailles de la machine.

Affichage du contenu des registres et de la mémoire

Pour afficher le contenu d'un registre — disons %ecx, il faut déjà se placer en mode pas à pas et puis utiliser la commande `print` :

```
[espoir.lifl.fr-sedoglav-/home/.../Sources] gdb -q sum
(gdb) break 27
Breakpoint 1 at 0x804807e: file sum.s, line 27.
(gdb) run
Starting program: /home/.../Sources/sum

Breakpoint 1, _start () at sum.s:27
27             movl $UnNom, %ecx  /* ECX va << pointer >> sur l'\'el\'ement
Current language:  auto; currently asm
(gdb) print $ecx
$1 = 0
(gdb) step
29      top:   addl (%ecx), %ebx
(gdb) print $ecx
$2 = 134516896
(gdb)  print /x $ecx
$3 = 0x80490a0
(gdb) printf "registre = %x en hexa et %d en decimale \n", $ecx, $ecx
registre = 80490a0 en hexa et 134516896 en decimale
(gdb) info register
eax            0x5      5
ecx            0x80490a0        134516896
edx            0x0      0
ebx            0x2b     43
esp            0xbffff710       0xbffff710
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048085        0x8048085
eflags         0x200306 2097926
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
```

```
gs             0x0      0
fctrl          0x37f    895
fstat          0x0      0
ftag           0xffff   65535
fiseg          0x0      0
fioff          0x0      0
foseg          0x0      0
fooff          0x0      0
fop            0x0      0
(gdb)
```

Remarquez que l'on peut aussi utiliser la commande `printf` qui nous permet d'afficher le contenu du registre comme en C (ici en hexadécimal dans une chaîne de caractères). La commande `info register` fournie la liste des registres ainsi que leurs contenus.

En utilisant les mêmes commandes, il nous est possible de déterminer le contenu du segment de données en mémoire à l'aide des labels :

```
(gdb) print UnNom
$4 = 43
(gdb) print &UnNom
$5 = (<data variable, no debug info> *) 0x80490a0
(gdb) print &UnNom
$6 = (<data variable, no debug info> *) 0x80490a0
(gdb) print *0x80490a0
$7 = 43
(gdb) print *(&UnNom)
$8 = 43
(gdb) print *(&UnNom+1)
$9 = 54
(gdb)
```

Le préfixe & permet d'avoir accès à l'adresse de l'objet et le préfixe * au contenu d'une adresse. On peut ainsi parcourir l'espace mémoire — notez que l'arithmétique est propre aux pointeurs i.e. additionner 1 à notre adresse consiste à additionner le nombre d'octets qui codent l'objet en mémoires (ici 4 octets).

Remarque 8. Dans notre code source, le label `UnNom` du segment de données pointait sur une suite de données de type différents (des long et une chaîne de caractères). Ce choix s'avère ici peu judicieux car :

```
(gdb) print *(&UnNom+2)
$10 = 23
(gdb) print *(&UnNom+3)
$11 = 32
(gdb) print *(&UnNom+4)
$12 = 76
(gdb) print *(&UnNom+5)
$13 = 1819043176
(gdb)
```

On obtient directement le bloc de données `hell` constitué de 4 octets sans pouvoir les distinguer directement. Il faut ruser un peu :

```
(gdb) print &UnNom+5
$13 = (<data variable, no debug info> *) 0x80490b4
(gdb) printf "%s\n",(&UnNom+5)
hello world
(gdb) printf "%c\n", *0x80490b4
h
(gdb) printf "%c\n", *0x80490b5
e
(gdb) printf "%c\n", *0x80490b6
```

Remarque 9. Plus généralement, il est possible d'obtenir l'affichage d'une zone mémoire grâce à la commande x :

```
(gdb) x /5dw 0x80490a0
0x80490a0 <UnNom>:      43      54      23      32
0x80490b0 <UnNom+16>:   76
(gdb) x /12cb 0x80490b4
0x80490b4 <UnNom+20>:   104 'h' 101 'e' 108 'l' 108 'l' 111 'o' 32 ' '  119 'w' 111 'o'
0x80490bc <UnNom+28>:   114 'r' 108 'l' 100 'd' 0 '\0'
```

Pour plus d'information sur cette commande, utilisez l'aide en ligne de gdb.

Dans le chapitre suivant, nous reviendrons sur la structure de la mémoire.

1.6 Exercices

Exercice 1 — Calcul de factorielle.

En utilisant l'exemple donné précédemment, écrire un programme qui calcule et stocke dans le registre %ebx la factorielle de 4. Les instructions arithmétiques nécessaires sont décrites dans l'annexe C.

Vérifier l'exécution de votre programme avec `gdb`.

Modifiez votre programme pour calculer la factorielle de 5 puis celle de 6. Que constatez vous concernant le stockage des résultats dans les registres.

Modifiez votre programme pour que la valeur dont il faut calculer la factorielle soit en mémoire (de données) et le résultat stocké en mémoire également.

Correction.

Nous avons utilisé dans ce programme l'instruction assembleur `jnz` sans la décrire. Ce type d'instruction de saut et la notion abstraite *d'instructions de contrôle* constituent le sujet de la section suivante.

Chapitre 2

Structures de contrôle

Dans notre contexte, on appelle *structure de contrôle* toute séquence d'instructions qui permet de contrôler la succession des actions d'un programme. En ce qui concerne l'assembleur, il s'agit exclusivement de séquence d'instructions de *sauts conditionnels*.

2.1 Organisation de la mémoire : segments et offsets

Sur une architecture 32 bits, la mémoire physique peut être considérée comme une suite d'octets. À chaque octet est associé un numéro de 0 à $2^{32}-1$; ainsi, on peut adresser environ 4 gigaoctets de mémoire.

Abstraction de la mémoire. La segmentation permet au programmeur de voir un processus en mémoire sous la forme de plusieurs espaces d'adressages : les segments. Ainsi, chaques *composantes* du programme (pile, code, tas, données, etc.) peut avoir son propre segment.

Par exemple, le code binaire du programme en cours d'exécution — stocké ou non en mémoire physique — peut être accessible par un segment. Dans cette abstraction, le code est vue comme une suite d'octets indicée de 0 à $2^{32} - 1$. En théorie, un segment peut donc recouvrir l'ensemble de la mémoire mais on n'utilise pas la totalité de cette dernière (le processeur permet de limiter l'accès à cet espace).

Mode d'adressage et notion de saut. Pour accéder à un octet, le processeur propose 3 mécanismes d'adressages :

- l'adressage direct (un registre 32 bits contient une adresse) ;
- l'adressage semi-direct (que nous n'expliciterons pas) ;
- l'adressage logique. On utilise pour ce faire une *adresse logique*. Cette adresse est composée du numéro du segment — contenu dans un registre 16 bits — et d'un offset — contenu dans un registre 32 bits.

C'est ce dernier mode d'adressage que nous utiliserons. Citons comme exemple d'adresse logique, l'adresse de la prochaîne instruction à exécuter :

– le registre %eip (Instruction Pointer) contient l'offset de la prochaîne instruction à exécuter ;
– le registre %cs (Code Segment) contient le numéro du segment mémoire dans lequel sont stocké les instructions assembleur du code à exécuter.

Effectuer un *saut* vers une instruction dans le flux d'instructions consiste à remplacer le contenu du registre %eip par l'offset de cette instruction. Pour illustrer ce point, nous allons décrire dans la section suivante une instruction effectuant cette opération.

2.2 Instruction de saut inconditionnelle `jmp`

Cette instruction provoque un saut à une adresse pouvant être spécifiée par différents types d'opérandes :

– une étiquette définie dans le segment de code. Remarquez que vous pourriez utiliser une étiquette définie dans un segment de données sans que le compilateur ne bronche : ce n'est qu'à l'exécution qu'une erreur surviendrait ;
– un registre généraliste. Si on utilise un registre 8 bits, le saut est qualifié de *court* puisqu'il ne peut modifier la valeur du registre %eip que d'au plus 127. L'usage est d'utiliser un registre de 16 bits : on obtient ainsi un saut *proche*. L'utilisation d'un registre de 32 bits correspond à un saut *lointain* généralement vers un autre segment de code ;
– une constante (cette possibilité est très fortement déconseillée).

Nous n'utiliserons que la première possibilité dans ce fascicule. Ainsi, dans le bout de code suivant :

```
done:
       movl $0,%ebx
       movl $1,%eax
       jmp fini           /* effectue un saut */
       movl $666,%ebx
       movl $666,%eax
fini:
       int  $0x80
```

les instructions :

```
       movl $666,%ebx
       movl $666,%eax
```

ne seront jamais exécutées du fait du saut inconditionnel `jmp fini`.

Exercice 2 — Saut inconditionnel.

Vérifiez cette assertion.

2.3 Instructions de saut conditionnelles

Ces instructions vérifient l'état d'un ou plusieurs drapeaux du registre EFLAGS (cf. la description de ce registre dans la section A.1). Si ce registre est dans l'état spécifié alors le saut défini par l'opérande de l'instruction est effectué. Sinon, l'instruction suivante est exécutée. Considérons l'exemple suivant :

```
top:   addl (%ecx), %ebx
       addl $4, %ecx      /* D\'eplace le << pointeur >> sur l'\'el\'ement
                             suivant                                         */
       decl %eax          /* D\'ecr\'emente le compteur EAX                 */
       jnz top            /* Si le contenu de EAX est non nul alors
                             ex\'ecuter le code \`a partir du label top     */
done:  movl %ebx, UnAutre /* sinon, le resultat est stock\'e                */
```

L'instruction `decl %eax` modifie le drapeau `ZF` en lui assignant la valeur 1 si le contenu de `%eax` est nul (ce comportement est indiqué dans la description de cette instruction dans l'annexe C).

L'instruction `jnz top` provoque un saut si, et seulement si, le drapeau `ZF` est positionné à 0. Cette instruction est appelée *jump if non zero* et en tire son nom. On obtient ainsi une structure itérative permettant la sommation dans le code ci-dessus.

Le tableau suivant regroupe les instructions conditionnelles de saut ainsi que la valeur des drapeaux correspondantes.

INSTRUCTION	**Conditions de saut**	**Indicateurs**
pour valeurs non signées		
JB/JNAE	below/ not above or equal	CF = 1
JAE/JNB	above or equal/not below	CF=0
JBE/JNA	below or equal	CF=1 et ZF=1
JA/JNBE	above/not below or equal	
JE/JZ	equal/zero	ZF=1
JNE/JNZ	not equal/not zero	ZF=0
pour valeurs signées		
JL/JNGE	less than/not greater than	SF=NOT OF
JGE/JNL	greater or equal/not less than	SF=OF
JLE/JNG	less or equal/not greater than	ZF=1 ou SF=NOT OF
JG/JNLE	greater than/not less or equal	ZF=0 ou SF=OF
JP/JPE	Parity even (paire)	PF=1
JNP/JNPO	Parity odd (impaire)	PF=0
JS	signé	SF=1
JNS	non signé	SF=0
JC	retenue	CF=1
JNC	pas de retenue	CF=0
JO	overflow	OF=1
JNO	pas d'overflow	OF=0

2.3.1 Conditionnelles

L'instruction `cmp` permet de comparer deux opérandes. Cette comparaison est basée sur la soustraction de la première opérande avec la seconde sans stocker le résultat. Le comportement des drapeaux est explicité dans l'annexe C dans la description de l'instruction `sub`.

Voici quelques exemples de structures conditionnelles classiques et leurs implantations en assembleur :

```
; si X>Y alors
;   <instructions>
; fsi
    mov X, reg1
    mov Y, reg2
    cmp reg2, reg1
    jna fsi1  ; not above
; <instructions>
fsi1:
```

```
; si X<Y alors
;   <instructions>
; fsi
      mov X, reg1
      mov Y, reg2
      cmp reg2, reg1
    jnb fsi2  ; not below
; <instructions>
fsi2:
```

```
; si X=Y alors
;   <instructions>
; fsi
    mov X, reg1
    mov Y, reg2
    cmp reg1, reg2
    jne fsi3
; <instructions>
fsi3:
```

```
; si X=0 alors
;   <instructions>
; fsi
      mov X, reg1
      and reg1, reg1   ; plus rapide que cmp reg1, 0
      jnz fsi4
; <instructions>
fsi4:
```

En combinant sauts conditionnel et inconditionnel, on peut mettre en place des structures plus complexes comme par exemple :

```
; si X=0 alors
;   <instructions_alors>
; sinon
;   <instruction_sinon>
; fsi
    mov X, AX
    and AX, AX
    jne sinon
; <instructions_alors>
    jmp fsi5
sinon:
; <instructions_sinon>
fsi5:
```

Exercice 3 — Affichage d'étoiles.

Le code assembleur ci-dessous affiche le caractère * grâce à un appel au noyau Linux :

```
.data

msg:
        .byte  42
.text

.globl _start

_start:
        movl $4,%eax        /* Le code d\'esignant la fonction \`a utiliser
                               (dans ce cas sys write) */
        movl $1,%ebx        /* 1 argument~: le num\'ero de la sortie
                               standard.                                  */
        movl $msg,%ecx      /* 2 argument~: un pointeur sur le message \`a
                               afficher.                                  */
        movl $1,%edx        /* 3 argument~: le nombre de caract\`ere(s) \`a
                               afficher.                                  */
        int  $0x80          /* Appel au noyau (interruption 0x80).        */

done:
        movl $1,%eax        /* Le code d\'esignant la fonction \`a utiliser
                               (dans ce cas sys exit) */
        movl $0,%ebx        /* 1 argument~: le code de retour du processus.  */
        int  $0x80          /* Appel au noyau (interruption 0x80).        */
```

On se propose d'afficher des étoiles dans les dispositions suivantes :

```
*****    *        *****    *****        *
*****    **        ****    ****        **
*****    ***        ***    ***        ***
*****    ****        **    **        ****
*****    *****        *    *        *****
```

Construisez un programme assembleur qui affiche successivement ces dispositions.

2.3.2 Boucles conditionnelles

Exercice 4 — Implantation itérative de l'algorithme d'Euclide.

On se propose de calculer le plus grand commun diviseur — pgcd — de deux entiers par l'algorithme d'Euclide. Cet algorithme repose sur le fait que, étant donné deux entiers a et b, le pgcd de a et de b est égal au pgcd de b et de r où r est le reste de la division euclidienne de a par b et ceci tant que r est différent de zéro. Le pgcd est alors le dernier diviseur utilisé.

Écrivez un programme assembleur qui, à partir de deux entiers stockés dans le segment de données, calcule leur pgcd (on suppose que ces entiers sont tous deux inférieurs à $2^{32} - 1$). Pour information, en assembleur le reste de la division euclidienne peut être obtenu par l'instruction `div` et la comparaison par l'instruction `cmp` (cf. annexe C).

Calculer le pgcd de 13481754 et de 1234715.

Correction.

2.3.3 Instructions de boucle énumérative : `loop`

L'instruction `loop` nécessite l'emploi des registres %ecx ou %cx comme compteurs. À chaque itération de boucle, le registre %ecx est automatiquement décrémenté. Si après décrémentation, %ecx est nul, on sort de la boucle pour exécuter l'instruction suivante ; sinon loop branche en fonction de son opérande (label ou saut court 8 bits). Par exemple, on peut écrire :

```
           movl $10,%ecx
           movl $0 ,%eax
DebutBoucle:
          add %ecx,%eax
          loop %ecx
```

pour faire la somme des entiers de 0 à 10 inclut.

Attention : si le registre — disons %cx — est nul au premier tour, il est décrémenté et sa valeur devient 65535. Dans ce cas de figure, on peut attendre un bon moment la sortie de boucle ! De même, le registre de boucle doit être modifié dans ce cadre avec grande prudence.

Exercice 5 — Mise en pratique.

Réécrivez le programme sum du premier chapitre 1.4.2 en utilisant une boucle `loop`.

On peut aussi utiliser les instructions décrites dans le tableau suivant :

Instruction	description
loop	décrémente CX et saut si ce registre est non nul
loope	décrémente CX et saut si ce registre est non nul et si ZF=1
loopz	décrémente CX et saut si ce registre est non nul et si ZF=1
loopne	décrémente CX et saut si ce registre est non nul et si ZF=0
loopnz	décrémente CX et saut si ce registre est non nul et si ZF=0

2.4 Exercices

Exercice 6 — Tri d'une suite d'entiers.

Construisez un programme assembleur qui étant donné une suite d'entiers définie dans le segment de donnée

```
Input:
       .word 24,7,274,2,504,1,13,149,4,81,44
```

fait le tri de cette suite et stocke le résultat dans une autre partie de ce même segment qui aura été réservée.

Correction.

Chapitre 3

Appels systèmes par le biais d'interruptions

3.1 Mécanisme d'interruption

La notion *d'interruption* se base sur un mécanisme par lequel certaines composantes physiques (horloge, E/S, mémoire, processeur) peuvent interrompre le traitement normal du processeur.

Une interruption est la commutation d'un processus à un autre provoquée par un signal du matériel.

Il existe différents types de signaux provoquant une interruption :

- logiciel : division par zéro, référence mémoire en dehors de l'espace autorisé au processus, dépassement de capacité (pile), appel système ;
- temporisateur : le processeur permet à l'OS d'effectuer régulièrement certaines fonctions (ordonnancement, mise à jours) ;
- E/S : signale l'achèvement normal d'une opération ou une erreur ;
- défaillance matérielle : coupure d'alimentation, erreur de parité mémoire.

Gestion des interruptions et vecteur des interruptions du pentium. Il existe un contrôleur d'interruption qui peut :

- masquer certaines interruptions dans des phases critiques du fonctionnement de l'unité centrale ;
- hiérarchiser les interruptions lorsque plusieurs d'entre elles interviennent en même temps (même modèle que l'ordonnancement).

Une partie de la gestion des interruptions est implantée directement dans le silicium sous forme d'une table. Une fois un signal détecté, il faut pouvoir déterminer la cause de l'interruption. Pour ce faire, on utilise 256 indicateurs qui pour le pentium sont :

0	erreur de division
1	exception de déverminage
2	interruption nulle
3	arrêt
4	dépassement interne détecté
5	dépassement de limite
6	opération code invalide
7	périphérique indisponible
8	faute sur type double
9	dépassement de segment coprocesseur
10	segment d'état de tâche invalide
11	segment non présent
12	faute de pile
13	protection générale
14	défaut de page
15	réservé par INTEL
16	erreur virgule flottante
17	contrôle d'alignement
18	contrôle machine
19-31	réservé INTEL
32-255	interruptions masquables

3.2 Interruptions et appels système du noyau Linux

Un processus peut provoquer une interruption. En assembleur, celle-ci est engendrée par l'instruction `int` suivie du numéro d'un service. Par exemple, pour exécuter une fonction du noyau Linux depuis un code assembleur, on peut utiliser le service `int $0x80`.

Une interruption est alors provoquée i.e. un code constitutif du noyau est exécuté. Pour mener à bien cette exécution, il faut de plus connaître :

– la fonction du noyau à exécuter ;
– comment transmettre les arguments à cette fonction ;
– comment récupérer la sortie de cette fonction.

Ces informations transitent par certains registres :

– la fonction à exécuter est définie par le contenu du registre %eax et la table 3.1 ;
– les arguments à transmettre sont déterminés par les registres %ebx, %ecx, %edx, %esi et %edi ;
– le code de retour de la fonction est stocké dans le registre %eax. De plus, certaines données passées en argument peuvent être modifiées par la fonction appelée.

%eax	Nom	Code source	%ebx	%ecx	%edx	%esi	%edi
1	sys_exit	kernel/exit.c	int				
2	sys_fork	arch/i386/kernel/process.c	strucl pt_regs				
3	sys_read	fs/read_write.c	unsigned int	char *	size_t		
4	sys_write	fs/read_write.c	unsigned int	char *	size_t		
5	sys_open	fs/open.c	const char *	int	int		
6	sys_close	fs/open.c	unsigned int				

Fig. 3.1 – Table des appels systèmes Linux

Dès la fin de la fonction appelée, l'exécution du programme assembleur reprend à l'instruction suivant l'interruption.

Un descriptif plus complet de ces appels système est disponible en annexe D.

3.3 Gestion d'entrée-sortie

Avant de manipuler des interruptions d'entrée-sortie, il nous faut présenter les notions de *descripteur de fichier* et de *numéro d'accès.*

- Un descripteur de fichier correspond à une chaîne de caractères ASCII d'au plus 128 octets terminée par un 0. Cette chaîne est composée d'un chemin d'accès et d'un nom de fichier. Ce nom est la seule information non optionnelle. Le descripteur peut être composé de caractères spéciaux (*, etc).
- Un numéro d'accès est associé par le système d'exploitation à chaque fichier. Ce numéro — codé sur 2 octets — permet d'identifier le fichier lors des manipulations. De plus, notons que certains périphériques sont gérés comme des fichiers ; ainsi, à l'entrée standard (usuellement le clavier) correspond le numéro d'accès 0 et on a les correspondances :

numéro	support
0	entrée standard
1	sortie standard (usuellement l'écran)
2	sortie d'erreur (idem)
3	sortie auxiliaire (usuellement un port)
4	sortie d'impression

3.3.1 Lecture et écriture

Exercice 7 — Entrée et sortie standard.

1. Écrire un programme qui lit au clavier un entier N codé sur un caractère (de 0 à 9 donc) et affiche une ligne de N caractères '`*`'.
2. Écrire un programme qui lit au clavier un entier N codé sur plusieurs caractères (saisie finie par "retour chariot" — code ASCII 13) et affiche une ligne de N caractères '`*`'.
3. Écrire un programme qui lit au clavier un entier N codé sur plusieurs caractères (saisie finie par "retour chariot") et affiche un carré composé de '`*`' de côté N.

3.3.2 Ouverture et fermeture d'un fichier

Exercice 8 — Manipulation de fichiers.

1. Créer un fichier `foo.essai` et remplissez le avec un descripteur de fichier `bar.essai`.
2. Après avoir créé un fichier contenant un descripteur de fichier, construisez un programme assembleur qui lit ce fichier et place le descripteur de fichier en mémoire.

3. En supposant que le descripteur de fichier précédent n'est pas associé à un fichier existant, construire un programme assembleur qui crée un fichier du même nom et stocke à l'intérieur la chaîne de caractères représentant le descripteur de fichier ainsi que le numéro d'accès (codé sur 2 octets).

Chapitre 4

Transferts et stockage de données

4.1 Transferts de données

4.1.1 L'instruction mov

Cette instruction s'utilise comme suit `mov source, destination`.

La destination et la source peuvent être représentées par des constantes, des registres, des étiquettes ou des *références mémoire*.

Nous avons déjà utilisé l'instruction `mov` avec des registres ou des étiquettes comme opérandes. Par exemple :

`mov label, %eax`	place le contenu de l'espace mémoire correspondant à l'étiquette `label` dans le registre %eax ;
`mov $label, %eax`	place l'adresse correspondant à l'étiquette `label` dans le registre %eax ;
`mov $0, %eax`	place la constante 0 dans le registre %eax.

Dans ce cadre, une référence mémoire détermine une adresse mémoire suivant la syntaxe :

registre de segment :déplacement(registre de base, registre de décalage, facteur de décalage)

Avant d'aller plus loin, rappelons qu'une adresse mémoire est spécifiée par deux entiers :

le numéro du segment dans lequel elle est définie. Au cours de l'exécution d'un code assembleur, Le registre %cs (Code Segment) contient le numéro du segment mémoire dans lequel sont stockées les instructions et le registre %ds (Data Segment) contient le numéro du segment de données ;

le déplacement relatif dans le segment correspondant à l'emplacement mémoire dans le segment.

La première composante d'une référence mémoire indique le segment mémoire considéré. La seconde composante indique un déplacement relatif égal à :

registre de base + registre de décalage * facteur de décalage + déplacement

Exemple. Pour mieux comprendre à quoi correspond cette syntaxe, considérons le code suivant :

```
.data
var:
    .long 1
.text
.globl _start

_start:
        movl    $var,%edx   /* cette commande pr\'epare la suite         */
        movl    $2,%eax     /* cette commande pr\'epare la suite         */

/* L'exemple a pour objectif de comprendre l'instruction suivante */

        movl    $0,%ds:-4(%edx,%eax,2)

        movl    $0,%ebx     /* Ces instructions permettent d'invoquer de */
        movl    $1,%eax     /* terminer l'ex\'ecution d'un programme     */
        int     $0x80       /* assembleur et sont indispensables         */
```

Explicitons à présent l'instruction `movl $0,%ds :-4(%edx,%eax,2)`. Elle place la constante 0 dans la mémoire à l'adresse spécifiée par :

le registre de segment : %ds dans le segment dont le numéro est dans le registre %ds i.e. dans le segment de données ;

le registre de base : %edx ce registre contient la base du déplacement relatif et dans notre cas l'offset de l'étiquette var.
À cette base s'ajoute un entier défini par

le registre de décalage : %eax — qui contient 2 dans notre exemple — multiplié par

le facteur de décalage — qui est 2 dans notre exemple — auquel pour finir on ajoute

le déplacement -4 dans notre cas.

Ce qui donne offset de var $+ 2 * 2 - 4 =$ offset de var.

Ainsi dans l'exemple ci-dessus, l'instruction `movl $0,%ds :-4(%edx,%eax,2)` est une façon compliquée d'écrire `movl $0,var`.

Remarque 10. Le facteur de décalage peut prendre les valeurs $1, 2, 4$ ou 8 ; s'il n'est pas spécifié, sa valeur par défaut est 1.

Les références mémoire peuvent présenter une syntaxe simplifiée, ainsi dans la référence valide :

`-4(%ebp)` seuls le registre de base et le déplacement sont spécifiés. Attention le registre de segment dans ce cadre est %ss comme indiqué dans l'annexe A.1 ;

`foo(,%eax,4)` le registre de segment non indiqué est %ds. Le segment de base est nul et on utilise le déplacement spécifié par l'étiquette `foo` en guise de base.

Exercice 9 — Exemples.

Considérons le code suivant :

```
.data
var:
    .string "ABCDEF"
dest:
    .space 7
.text
.globl _start

_start:
        movl    var,%eax
        movl    $0,%eax
        movl    $0,%edx
        movb    var,%al
        movb    var+2,%al
        movl    $var,%eax
        movl    $var,%edx
        movl    $2,%eax
        movl    $0,%ds:-3(%edx,%eax,3)

        movl    $0,%ebx     /* Ces instructions permettent d'invoquer de */
        movl    $1,%eax     /* terminer l'ex\'ecution d'un programme     */
        int     $0x80       /* assembleur et sont indispensable         $*/
```

Pour chaque ligne, indiquez le résultat produit.

Exercice 10 — Manipulation de tableau : multiplication de matrices.

Construisez un programme qui effectue la multiplication des matrices A et B définies dans le segment de données par :

```
.data
A:
   .long 97,24,34,45,12
   .long 56,23,78,24,39
   .long 45,42,10,94,93
   .long 67,84,31,94,86
   .long 73,54,59,47,98
B:
   .long 73,23,98,74,06
   .long 54,76,34,98,73
   .long 75,04,74,18,83
   .long 34,89,73,38,55
   .long 13,84,73,54,73
C:
   .space 100
```

4.1.2 Conversion de données

Il est possible d'augmenter la taille du codage des données entières de 2 manières différentes en utilisant :

– des instructions spécifiques ;
– des instructions classiques suffixées.

Instructions de conversion L'architecture Intel propose les instructions de conversion suivante :

cbtw – étend un octet signé contenu dans %al en un mot signé contenu dans %ax ;
cwtl – étend un mot signé contenu dans %ax en un long signé contenu dans %eax ;
cltd – étend un long signé contenu dans %eax en un double long signé contenu dans %edx :%eax.

Suffixes de conversion Plus généralement, l'instruction `mov` admet des suffixes à 2 niveaux indiquant une conversion.

Le premier niveau indique si on étend la taille de codage en conservant le signe (`movs...`) ou en étendant le codage par des 0 (`movz...`).

Le second niveau indique la taille de départ et celle d'arrivée :

bl	de octet à long,	bq	de octet à quadruple mot,
bw	de octet à mot,	wq	de mot à quadruple mot,
wl	de mot à long,	lq	de long à quadruple mot.

Ainsi l'instruction `movzbl %dl,%eax` déplace l'octet contenu dans le registre %dl dans le registre %eax en le complétant par des 0.

4.2 Manipulation de la pile

Schématiquement, une pile est une structure de données linéaire pour laquelle les insertions et les suppressions d'éléments se font toutes *du même coté*. On parle de structure LIFO : Last In First Out.

Plus formellement, on peut considérer un ensemble d'éléments E et noter Pil(E) l'ensemble de toutes les piles sur E. Par exemple, les entiers peuvent constituer l'ensemble E ; la pile vide P_0 est dans Pil(E). Les opérations usuelles sur une pile sont :

– estVide est une application de Pil(E) dans (vrai, faux), estVide(P) est vrai si, et seulement si, P est la pile P_0.
– empiler est une application de $E \times$ Pil(E) dans Pil(E).
– depiler est une application de Pil(E) $\setminus P_0$ dans Pil(E).
– supprimer est une application de Pil(E) $\setminus P_0$ dans Pil(E).

Si x est un élément de E, les relations satisfaites par une pile P et ces opérations sont :

1. estVide(P_0) = vrai
2. supprimer(empiler(x, P)) = P
3. estVide(empiler(x, P)) = faux
4. depiler(empiler(x, P)) = x

Cette dernière règle caractérise les piles.

4.2.1 Implantation de la pile

Un segment de la mémoire est dévolu à la pile.

Les registres SS et SP sont deux registres servant à gérer la pile :

SS (Stack Segment i.e. segment de pile) est un registre 16 bits contenant l'adresse du segment de pile courant ;

ESP (Stack Pointer i.e. pointeur de pile) est le déplacement pour atteindre le sommet de la pile.

Ainsi, SP pointe sur le dernier bloc de mémoire occupé de la pile.

L'assembleur vous fera manipuler une pile qui est stockée “en fond de panier”, c.à.d dans les adresses les plus hautes de la mémoire. Ainsi, la base de la pile se trouve à l'adresse maximale, et elle s'accroît vers les adresses basses. A l'inverse, les programmes se développent sur le “tas”, en bas de mémoire, juste au dessus de la zone réservée pour le système. Pile et Tas croissent donc à l'inverse l'un de l'autre.

4.2.2 Manipulation de la pile

Les modification de la structure de la pile se font par les instructions :

- `push reg` (empiler depuis le registre reg). Lorsque l'on empile un élément sur la pile, l'adresse contenue dans SP est décrémentée de 4 octets (car un emplacement de la pile fait 32 shannons). En effet, lorsque l'on parcourt la pile de la base vers le sommet, les adresses décroissent.
- `pop reg` (dépiler vers le registre reg). Cette instruction incrémente de 4 octets la valeur de SP. Attention, lorsque la pile est vide SP pointe sous la pile (l'emplacement mémoire en-dessous de la base de la pile) et un nouveau pop provoquera une erreur.

Il est aussi possible — pour lire ou modifier des valeurs dans la pile — d'utiliser les références mémoire.

Exercice 11 — Implantation d'une calculette élémentaire.

On se propose de réaliser une calculette qui prend en entrée une chaîne de caractères représentant une expression arithmétique en notation postfixe et qui affiche le résultat de l'évaluation de cette expression. Par exemple, on désire fournir à notre programme — par le biais de l'entrée standard — la chaîne de caractères `3 5 + 4 *` et que notre programme affiche `32`.

Format des entrées. Nous nous limitons à l'alphabet $\{*, +, -, 0, \ldots, 9\}$ pour faire simple. Pour des expressions arithmétiques sur cet alphabet comme par exemple :



on dispose de plusieurs codages possibles sous forme d'une chaîne de caractères :

- préfixe : $*, +, 1, 7, +, 2, 4$
- infixe : $1, +, 7, *, 2, +, 4$
- postfixe : $1, 7, +, 2, 4, +, *$

Dans cette exercice, nous allons utilisez ce dernier codage.

Principe de l'évaluation. Pour calculer une expression arithmétique codée en postfixe par une chaîne de caractères, il suffit de parcourir cette dernière et :

- d'empiler les entiers que l'on rencontre ;
- lorsque l'on rencontre un opérateur, il nous faut
  - dépiler les 2 derniers entiers ;
  - leur appliquer l'opérateur ;
  - empiler le résultat ;
- une fois le parcours terminé, le résultat du calcul devrait être dépilé et affiché.

Exercice 12 — Convertion de la notation infixe à la notation postfixe.

On se propose de compléter l'exercice précédant de manière à ce que l'entrée soit une chaîne ce caractères représentant le codage infixe d'une expression arithmétique (on s'autorise l'utilisation des parenthèses).

Pour ce faire, il nous faut convertir ce codage infixe en codage postfixe. Pour implanter notre première étape, on va s'aider d'un automate. Il s'agit d'une représentation d'un algorithme composée d'états et de transitions. Dans la figure 4.1, les transitions correspondent aux flèches du diagramme et les états aux encadrés. Dans le programme assembleur associé, les transitions seront implantées par des sauts et les états par des bouts de code. On suppose que la formule de départ est stockée en mémoire sous la forme d'une chaîne de caractères codée par le standard ASCII ; la fin de la chaîne est représentée par une virgule `,`. On dispose d'une pile et d'une adresse `RES` où le résultat sera stocké à la fin de la conversion. Explicitons maintenant la fonction de chaque état :

- La `récupération de caractère` consiste à chercher en mémoire un caractère et à incrémenter un pointeur afin de considérer le suivant dans la chaîne de départ ;
- l'`analyse et la décision` consiste à, soit ranger un chiffre — après conversion — dans `RES`, soit à aiguiller l'exécution du programme vers un des autres états suivant la nature des caractères récupéré et contenu dans le sommet de la pile (voir la suite et la figure 4.2) ;
- l'état `empiler` consiste à empiler le caractère récupéré ;
- l'état `dépiler` consiste à dépiler le caractère récupéré et à décrémenter le pointeur utilisé pour la récupération ;
- l'état `dépiler vers RES` consiste à dépiler un caractère et à le stocker dans `RES` puis à empiler le caractère récupéré ;
- l'état `erreur` indique que la construction de la chaîne de départ est défectueuse ;
- l'état `fin` marque la fin du processus de conversion.

Le plus gros du travail consiste à bien comprendre les actions réalisées dans l'état `analyse et la décision`. Nous allons donc expliciter certains points.

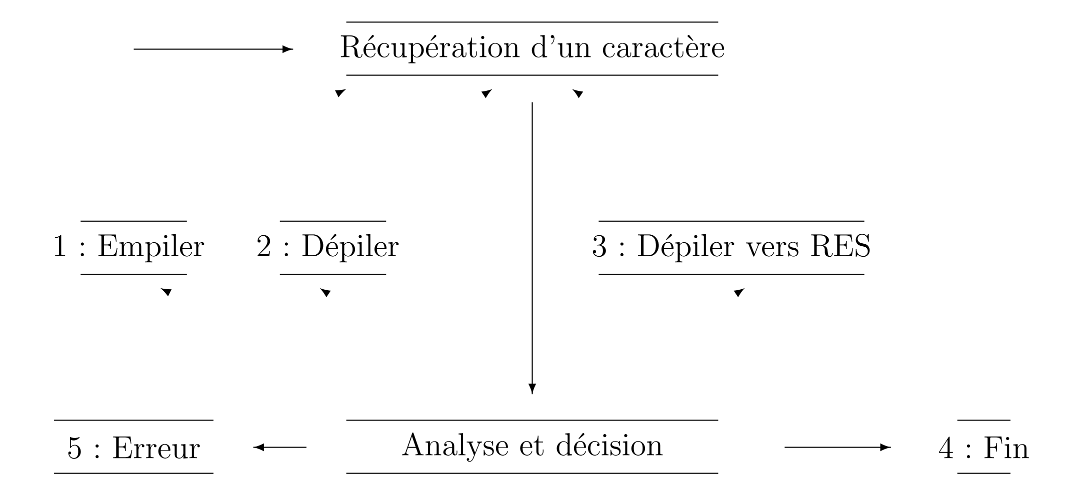


FIG. 4.1 – Automate représentant un algorithme de convertion de l'infixe vers le postfixe

Quand un caractère est récupéré dans la mémoire, il est soit empilé soit stocké dans `RES`. Un chiffre est toujours stocké dans la mémoire. Par contre, si le caractère est un symbole (+,*, etc.), l'action a effectuée dépend de ce que contient la pile. Considérons le cas où la chaîne de départ est $1 + 1 + 1, ;$ les tableaux suivants (1) représente à chaque étape la mémoire contenant la chaîne d'entrée, (2) représente la pile et (3) la mémoire contenant le résultat :

1														
+			+											
1			1			1								
+			+			+			+					+
1			1			1			1		1	1		1
,			,		1	,	+	1	,	+	1	,	+	1
(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)

					+
		1			1
		+			+
		1			1
,	+	1		,	1
(1)	(2)	(3)	(1)	(2)	(3)

(	40	+	43
)	41	,	44
$\star$	42	0	48
		9	57

Le tableau suivant indique les transitions effectuées à partir de l'état `analyse et décision`. Il est malheureusement incomplet ; n'y figure que les transitions déduites des tableaux

précédents.

		caractère récupéré					
		,	+	*	(	)	vide
sommet de la pile	,	5	5	5	5	5	4
	+	3	3	3	1	3	5
	⋆	3	3	3	1	3	5
	(	2	1	1	1	1	5
	)	2	1	1	1	1	5
	vide	4	1	1	1	5	5

FIG. 4.2 – Table de décisions utilisées dans l'état `analyse et décision`

Questions.

1. Construisez un programme qui prend en entrée une chaîne ce caractères représentant le codage infixe d'une expression arithmétique (on s'autorise l'utilisation des parenthèses) et qui affiche le codage postfixe correspondant.
2. Modifiez le programme de l'exercice précédent afin que l'entrée soit un codage infixe.
3. Modifiez vos programmes afin que l'on puisse manipuler des nombres et non plus des chiffres.

Chapitre 5

Appel de fonction et passage d'argument

Annexe A

Architecture de type Intel 32 bits

A.1 Les registres d'un processeur 32 bits de type Intel

Pour plus d'information sur ce sujet, nous renvoyons au manuel [1]. Un processeur 32 bits de type Intel dispose de 16 registres qui sont classables en trois catégories :

- les 8 registres généralistes utilisés pour contenir des données et des pointeurs ;
- les registres de segments utilisés pour contenir l'adresses des différents segments (données, code, etc.) ;
- les registres de contrôle et de statut qui donnent des informations sur l'exécution du programme en cours.

Le registre d'accumulation %eax

Ce registre généraliste peut contenir les données des utilisateurs.

Remarque 11. Ce registre joue le rôle d'opérande implicite dans de nombreuses opérations : `mul`, `div`, etc. Dans ce cas le résultat de l'opération est stocké dans ce même registre.

Le registre %ebx (base index) utilisé comme pointeur sur les données

Usuellement, ce registre contient un offset relatif au segment de données permettant de repérer une information de ce segment.

Le registre ECX utilisé comme compteur

Ce registre est utilisé pour les boucles et les opérations sur les chaînes de caractères.

Le registre EDX utilisé pour les entrées/sorties

Ce registre est généralement utilisé pour contenir des offsets lors des opérations d'affichage ou de saisie par exemple.

Le registre EIP (Instruction Pointer)

Le registre EIP contient l'offset de la prochaîne instruction à exécuter. Il est modifié automatique à chaque exécution et peut être manipulé par des instruction du type `jmp`, `call`, `ret`, etc. On ne peut pas accéder directement à ce registre.

Le registre EDI (Destination Index) et le registre ESI (Source Index)

Ces registres sont utilisés lors des opérations sur les manipulations — copie, etc. — de suites d'octets.

Le registre CS (Code Segment)

Ce registre 16 bits contient le numéro du segment mémoire dans lequel sont stockées les instructions assembleur du code à exécuter. On ne peut pas accéder directement à ce registre.

Le registre SS (Stack Segment)

Ce registre 16 bits contient le numéro du segment mémoire dans lequel est stockée la pile. On peut accéder directement à ce registre ce qui permet d'utiliser plusieurs piles.

Le registre ESP (Stack Pointer)

Ce registre 32 bits contient le déplacement pour atteindre le sommet de la pile. La partie basse 16 bits de ce registre peut être utilisée comme le registre SP.

Le registre EBP (Frame Base Pointer)

Ce registre 32 bits contient un déplacement correspondant à une position dans la pile. Ce registre sert à pointer sur une donnée dans la pile. La partie basse 16 bits de ce registre peut être utilisée comme le registre BP.

Les registres DS, ES, FS, GS (Data Segment)

Ces registres 16 bits contiennent les numéros de segment mémoire dans lesquels sont stockées des données. Le registre DS est utilisé pour le segment de données du code ; les autres permettent de référencer d'autres types de données. On ne peut pas accéder directement à ce registre.

Le registre EFLAG

Ce registre est structuré comme suit :

EFLAG

31	16	15	0
		FLAG	

Les bits $1, 3, 5, 15$ et les bits de 22 à 31 sont réservés par le microprocesseur pour son fonctionnement interne. Les 21 autres bits forment des drapeaux i.e. des indicateurs et nous ne nous intéresserons qu'au sous registre FLAG. Il existe quatre actions possibles d'une instruction sur les drapeaux :

- \- le drapeau n'est pas modifié ;
- ? on ne peut rien dire sur l'état de l'indicateur après l'instruction ;
- * le drapeau sera modifié par l'instruction et cette modification nous renseignera sur le déroulement de l'instruction ;
- 0 ou 1 valeur du drapeau après l'instruction.

Les fonctions des drapeaux sont résumées dans le tableau ci-dessus :

21	ID	Identification Flag	
20	VIP	Virtual Interrupt Pending	
19	VIF	Virtual Interrupt Flag	
18	AC	Alignment check	
17	VM	Virtual 8086 Mode	
16	RF	Resume Flag	
14	NT	Nested Task	Mis à 1 lorsque la tâche courante est associée à la tâche précédente, 0 sinon.
12 & 13	IOPL	I/O Privilege Level	Contient un nombre qui indique le niveau de privilège du programme en cours d'exécution
11	OF	Overflow Flag	Mis à 1 si une opération provoque un dépassement de capacité
10	DF	Direction Flag	Fixe le sens dans lequel seront effectuées les opérations de traitement de chaînes de caractères. STD le met à 1, CLD à 0.
9	IF	Interrupt Enable Flag	Mis à 1, il autorise les interruptions. S'il vaut 0, il les empêche.
8	TF	Trap Flag	Mis à 1, il force le processeur à fonctionner pas à pas.
7	SF	Sign Flag	Prend la valeur du bit de poids fort de l'accumulateur après un calcul.
6	ZF	Zero Flag	Mis à 1 si le résultat d'un calcul (i.e. le contenu de l'accumulateur) vaut 0.
4	AF	Auxiliary Carry Flag	Mis à 1 si un calcul en BCD non compacté produit une retenue.
2	PF	Parity Flag	Mis à 1 si les 4 bits de poids faible du résultat contiennent un nombre pair de 1.
0	CF	Carry Flag	Mis à 1 si un calcul produit une retenue.

Annexe B

Descriptif des directives

Toutes les directives commencent par un point.

B.1 Les directives indiquant une section

.data : Segment de données

Syntaxe :
.data

Description : Cette directive indique le début d'un segment contenant des données accessible en lecture et en écriture. Celui-ci est défini lors de la compilation ; son contenu est modifiable lors de l'exécution et il est généralement destiné aux variables globales.

.rodata : Segment de données en lecture seule

Syntaxe :
.rodata

Description : Cette directive indique le début d'un segment contenant des données uniquement accessible en lecture. Celui-ci est défini lors de la compilation ; son contenu est modifiable lors de l'exécution et il est généralement destiné aux variables globales.

.bss : Segment de stockage de données

Syntaxe :
.bss

Description : Cette directive indique le début d'un segment. Ce segment n'est défini que lors de l'exécution de votre programme et contient des octets initialisés à 0. Il est destiné à contenir des variables non initialisées et/ou fournir de l'espace mémoire de stockage.

.text : Segment de code

Syntaxe :
.text

Description : Cette directive indique le début d'un segment. Celui-ci est défini lors de l'assemblage et contient le code à exécuter. Bien qu'on puisse le modifier dynamiquement, c'est vivement déconseillé. De plus, ce segment peut être partagé entre plusieurs processus.

B.2 Indications sur la taille du codage

.byte : Déclaration d'un nombre sur 1 octet

Syntaxe :
.byte NB ou .byte NB1,...,NBn

Description : Pour chaque nombre NB, ce nombre est codé sur 1 octet à l'exécution.

.long : Déclaration d'un nombre sur 4 octets

Syntaxe :
.long NB ou .long NB1,...,NBn

Description : Pour chaque nombre NB, ce nombre est codé sur 4 octets à l'exécution.

.zero : Réservation d'un nombre d'octets tous mis à 0

Syntaxe :
.zero NB

Description : On réserve en mémoire NB octets tous mis à 0.

.space : Réservation d'un nombre spécifié d'octets ayant une valeur donnée

Syntaxe :
.space SIZE, FILL

Description : Cette directive réserve SIZE octets et leurs assigne la valeur FILL.

.word : Déclaration d'un nombre sur 2 octets

Syntaxe :
.word NB ou .word NB1,. . .,NBn

Description : Pour chaque nombre NB, ce nombre est codé sur 2 octets à l'exécution.

B.3 Déclaration et porté des symboles

Un code contient des symboles i.e. des étiquettes permettant de le structurer. Ces symboles peuvent être accessibles depuis d'autres code ou non.

.local : Déclaration locale d'une étiquette

Syntaxe :
.local `label`

Description : Indique qu'une étiquette ne peut être utilisée que dans le code la contenant.

.global : Déclaration globale d'une étiquette

Syntaxe :
.global `label`

Description : Indique qu'une étiquette est accessible à l'édition de liens afin d'être utilisable dans d'autre code que celui la contenant

.comm : Déclaration d'un symbol commun

Syntaxe :
.comm `symbol`, `taille`

Description : Cette directive permet de déclarer un symbole qui n'est pas défini au moment de la compilation. Lors de l'édition de liens, si l'éditeur de liens ne trouve pas de définition pour ce symbole, `taille` octets seront réservés dans le segment de données.

Annexe C

Descriptif des instructions

Pour chaque instruction, nous présentons :

- la syntaxe permettant l'utilisation des différentes dérivées de cette instruction ;
- une description de cette instruction ;
- l'effet de cette dernière sur le registre drapeaux (principalement pour les instructions relatives à l'unité arithmétique) ;
- les différents types d'opérandes possibles. Ainsi,
  idata représente une constante ;
  reg représente un registre ;
  mem représente une étiquette associée à un espace mémoire ou une adresse.

C.1 Arithmétique classique

add : addition d'entiers

Syntaxe gas ATT et opérandes possibles :

			source	destination
add	src	dest	idata	reg
addb	src	dest	idata	mem
addl	src	dest	reg	reg
addw	src	dest	mem	reg
			reg	mem

Description : Cette instruction additionne le contenu de l'opérande src avec celui de l'opérande dest et stocke le résultat dans l'espace désigné par l'opérande dest. Les opérandes doivent représenter des objets de même taille en bits. De plus, si les opérandes sont
– des entiers signés, le drapeau OF indique un résultat invalide ;
– des entiers non signées, le drapeau CF indique une retenue.

Drapeaux :

OF	DF	IF	TF	SF	ZF		AF		PF		CF
*	-	-	-	*	*	-	*	-	*	-	*

cmp : compare deux opérandes

Syntaxe gas ATT et opérandes possibles :

cmp	op1	op2
cmpb	op1	op2
cmpl	op1	op2
cmpw	op1	op2

op1	op2
idata	reg
idata	mem
reg	reg
mem	reg
reg	mem

Description : Cette instruction compare `op1` à `op2` et positionne des drapeaux du registre EFLAGS suivant le résultat :

Condition	Comparaison signée	Comparaison non signée
op1 ¡ op2	ZF == 0 et SF == OF	ZF == 0 et CF == 0
op1 ¡= op2	SF == OF	CF == 0
op1 == op2	ZF == 1	ZF == 1
op1 ¿= op2	ZF == 1 ou SF != OF	ZF == 1 ou CF == 1
op1 ¿ op2	SF != OF	CF == 1

Drapeaux :

OF	DF	IF	TF	SF	ZF		AF		PF		CF
*	-	-	-	*	*	-	*	-	*	-	*

dec : décrémentation

Syntaxe gas ATT et opérandes possibles :

dec	op
decb	op
decw	op
decl	op

op
reg
mem

Description : Cette instruction soustrait 1 à op.
Drapeaux :

OF	DF	IF	TF	SF	ZF		AF		PF		CF
*	-	-	-	*	*	-	*	-	*	-	-

div : division d'entiers non signés

Syntaxe gas ATT et opérandes possibles :

			op
div	op		reg
divb	op		mem
divw	op		
divl	op		

Description : Si l'opérande de cette instruction est codée sur 8 bits, le contenu du registre AX est divisé par l'opérande, le quotient est stocké dans AL et le reste dans AH.
Si l'opérande de cette instruction est codée sur 16 bits, l'entier défini par la paire de registre DX :AX est divisé par l'opérande, le quotient est stocké dans AX et le reste dans DX. Dans la pair de registre DX :AX, le poids faible est AX.
Si l'opérande de cette instruction est codée sur 32 bits, l'entier défini par la pair de registre EDX :EAX est divisé par l'opérande, le quotient est stocké dans EAX et le reste dans EDX.
Le débordement n'est pas indiqué par le drapeau CF.
En cas de résultat non entier, une troncation à 0 est faite.
Drapeaux :

OF	DF	IF	TF	SF	ZF		AF		PF		CF
?	-	-	-	?	?	-	?	-	?	-	?

inc : incrémentation

Syntaxe gas ATT et opérandes possibles :

			op
dec	op		reg
decb	op		mem
decw	op		
decl	op		

Description : Cette instruction additionne 1 à op.
Drapeaux :

OF	DF	IF	TF	SF	ZF		AF		PF		CF
*	-	-	-	*	*	-	*	-	*	-	-

mul : multiplication non signée

Syntaxe gas ATT et opérandes possibles :

mul	src
mulb	src
mulw	src
mull	src

src
reg
mem

Description : Cette instruction multiplie le contenu de l'accumulateur EAX (AX, AL suivant la taille de src) avec src. Le résultat est placé dans l'accumulateur.
Le résultat est calculé à concurrence du double de la taille de src. Par exemple,

– si src est codée sur 1 octet, la multiplication est faite avec AL et le résultat est codé sur 2 octets dans AX ;
– si src est codée sur 2 octets, la multiplication est faite avec AX et le résultat est codé sur 4 octets dans la paire de registre DX :AX ;
– si src est codée sur 4 octets, la multiplication est faite avec EAX et le résultat est codé sur 8 octets dans la paire de registre EDX :EAX.

Les drapeaux OF et CF sont mis à 0 si la moitié supérieure du résultat est nulle et à 1 sinon.
Drapeaux :

OF	DF	IF	TF	SF	ZF		AF		PF		CF
*	-	-	-	?	?	-	?	-	?	-	*

sub : soustraction d'entiers

Syntaxe gas ATT et opérandes possibles :

			source	destination
sub	src	dest	idata	reg
subb	src	dest	idata	mem
subl	src	dest	reg	reg
subw	src	dest	mem	reg
			reg	mem

Description : Cette instruction soustrait le contenu de l'opérande src avec celui de l'opérande dest et stocke le résultat dans l'espace désigné par l'opérande dest. Les opérandes doivent représenter des objets de même taille en bits. De plus, cette instruction provoque un double calcul : le processeur fait simultanément la soustraction une fois en supposant les données signées et une fois en les supposant non signées.
Ainsi, le drapeau OF est levé si un débordement survient lors de la soustraction signée et le drapeau CF est levé si une retenue survient lors de la soustraction non signée. Le drapeaux CF indique le signe du résultat si nécessaire.
Drapeaux :

OF	DF	IF	TF	SF	ZF		AF		PF		CF
*	-	-	-	*	*	-	*	-	*	-	*

C.2 Déplacement des données

mov : déplacement de données

Syntaxe gas ATT et opérandes possibles :

mov	src	dest
movb	src	dest
movw	src	dest
movl	src	dest

source	destination
idata	reg
idata	mem
reg	reg
mem	reg
reg	mem

Description : Cette instruction copie le contenu de l'opérande src dans l'opérande dest.
Drapeaux :

OF	DF	IF	TF	SF	ZF		AF		PF		CF
-	-	-	-	-	-	-	-	-	-	-	-

pop : dépilement de données

Syntaxe gas ATT et opérandes possibles :

pop	src
popw	src
popl	src

src
reg
mem

Description : Cette instruction incrémente de 4 octets la valeur de SP. Attention, lorsque la pile est vide SP pointe sous la pile (l'emplacement mémoire en-dessous de la base de la pile) et un nouveau pop provoquera une erreur.
Drapeaux :

OF	DF	IF	TF	SF	ZF		AF		PF		CF
-	-	-	-	-	-	-	-	-	-	-	-

push : empilement de données

Syntaxe gas ATT et opérandes possibles :

		dest
push	dest	idata
pushw	dest	reg
pushl	dest	mem

Description : Lorsque l'on empile un élément sur la pile, l'adresse contenue dans SP est décrémentée de 4 octets (car un emplacement de la pile fait 32 shannon de longueur). En effet, lorsque l'on parcourt la pile de la base vers le sommet, les adresses décroissent.
Drapeaux :

OF	DF	IF	TF	SF	ZF		AF		PF		CF
-	-	-	-	-	-	-	-	-	-	-	-

C.3 Sauts

jmp : saut inconditionnel

Syntaxe gas ATT et opérandes possibles :

		dest
		reg
mov	dest	mem
		idata
		offset

Description : Cette instruction effectue un saut inconditionnel. Elle ne modifie pas les drapeaux courant indiqués ci-dessous. L'adresse destination peut être spécifiée de manière absolue — une adresse stockée dans un registre, une mémoire — ou relative — une valeur immédiate éventuellement négative.
Drapeaux :

OF	DF	IF	TF	SF	ZF		AF		PF		CF
-	-	-	-	-	-	-	-	-	-	-	-

C.4 Appel de fonction

leave : Instruction précédent la fin d'une fonction en C

Syntaxe gas ATT et opérandes possibles :

leave

Description : Cette instruction copie le contenu du registre de cadre %ebp dans le registre de pile %esp puis dépile la pile à destination du registre %ebp. Cette instruction précède généralement l'instruction `ret`.
Drapeaux :

OF	DF	IF	TF	SF	ZF		AF		PF		CF
-	-	-	-	-	-	-	-	-	-	-	-

call : Appel de fonction

Syntaxe gas ATT et opérandes possibles :

call dest

dest
reg
mem
idata
offset

Description : Cette instruction empile l'adresse de l'instruction qui la suit et effectue un saut à l'adresse dest. L'adresse destination peut être spécifiée de manière absolue — une adresse stockée dans un registre, une mémoire — ou relative — une valeur immédiate éventuellement négative. Par exemple, l'instruction `call .+0x3` effectue un branchement à une instruction se trouvant 3 octets plus loin dans le segment de code.
Drapeaux :

OF	DF	IF	TF	SF	ZF		AF		PF		CF
-	-	-	-	-	-	-	-	-	-	-	-

C.5 Misc

lea : Load Effective Address

Syntaxe gas ATT et opérandes possibles :

lea src dest

src	dest
mem	reg

Description : Cette instruction calcul l'adresse de la première opérande et range le résultat dans la seconde.
Drapeaux :

OF	DF	IF	TF	SF	ZF		AF		PF		CF
-	-	-	-	-	-	-	-	-	-	-	-

Annexe D

Descriptif des appels système

D.1 Entrée – sortie

sys_read : Lecture depuis un flux

Description : Cette fonction récupère %edx octets depuis le support de numéro %ebx et les place à partir de l'adresse %ecx. Cette copie est interrompue dés qu'un retour chariot est rencontré.
Entrée(s) :

%eax	3
%ebx	le numéro d'accés du flux
%ecx	un pointeur sur l'espace où stocker les données lues
%edx	le nombre d'octets à prendre en compte

Sortie(s) :
%eax contient le nombre d'octets lu.
Prototype C :
```
asmlinkage ssize_t sys_read(unsigned int fd, char __user *
buf, size_t count)
```

sys_write : Écriture dans un flux

Description : Cette fonction écrit %edx octets sur le support de numéro %ebx en les stockant à partir de l'adresse %ecx.

Entrée(s) :

%eax	4
%ebx	le numéro d'accés du flux
%ecx	un pointeur sur l'espace contenant les données à écrire
%edx	le nombre d'octets à prendre en compte

Sortie(s) :

%eax	contient le nombre d'octets écrit.

Prototype C :

```
asmlinkage ssize_t sys_write(unsigned int fd, char __user *
buf, size_t count)
```

sys_open : Ouverture d'un flux

Description : Cette fonction ouvre un flux — un fichier — de nom filename et retourne l'identificateur assocé.

Entrée(s) :

%eax	5
%ebx	un pointeur sur la chaîne de caractère représentant le nom du fichier
%ecx	des drapeaux (00 read-only, 01, write only, 10 read-write, 11 special)
%edx	le mode (disons 0 sans plus de détails).

Sortie(s) :

%eax	contient l'identificateur associé au fichier.

Prototype C :

```
asmlinkage long sys_open(const char __user * filename, int
flags, int mode)
```

sys_close : Fermeture d'un flux

Description : Cette fonction ferme le flux d'identificateur fd.

Entrée(s) :

%eax	6
%ebx	l'identificateur du flux.

Sortie(s) :

Prototype C :

```
asmlinkage long sys_close(unsigned int fd)
```

D.2

sys_exit : Fin de processus

Description : Cette fonction permet de terminer le processus courant en retournant l'entier `error_code` au processus père.

Entrée(s) :

%eax	1
%ebx	un entier `error_code`

Sortie(s) :

%eax	contient le nombre d'octets lu.

Prototype C :

```
asmlinkage long sys_exit(int error_code)
```

Annexe E

Récapitulatif des commandes gdb

Lancement et arrêt de gdb

gdb fichier	:	lancement de l'environement gdb
quit	:	sortie de l'environement gdb

Remarque. Le raccourci clavier `CTRL-C` ne provoque pas la terminaison de `gdb` mais interrompt la commande courante.

Commandes générales

run	:	lancement d'un programme dans l'environement gdb
kill	:	arrêt définitif d'un programme

Manipulation des points d'arrêt

break FCT	:	placer un point d'arrêt au début de la fonction FCT
break *ADDR	:	placer un point d'arrêt à l'adresse ADDR
break NUML	:	placer un point d'arrêt à la ligne NUML
disable NUM	:	inactive le point d'arrêt NUM
enable NUM	:	réactive le point d'arrêt NUM
delete NUM	:	supprime le point d'arrêt NUM
delete	:	supprime tous les point d'arrêts

Exécution d'un programme pas à pas

step	:	exécute une instruction élémentaire
step NUM	:	exécute NUM instructions élémentaires
next	:	exécute une instruction (y compris les fonctions appelées)
next NUM	:	exécute NUM instructions (y compris les fonctions appelées)
until LOC	:	exécute les instructions jusqu'à ce que LOC soit atteint
continue	:	reprend l'exécution
continue NUM	:	reprend l'exécution en ignorant les points d'arrêt NUM fois
finish	:	exécute jusqu'à ce que la fin de la fonction en cours
where	:	affiche la position actuelle

Affichage du code et des données

list	:	affiche le code source par paquet de 10 lignes
list NUML	:	affiche le code source à partir de NUML
disas	:	affiche le code autour de la position courante
disas ADDR	:	affiche le code autour l'adresse ADDR
disas ADDR1 ADDR2	:	affiche le code entre les adresses ADDR1 et ADDR2
print $REG	:	affiche le contenu du registre REG
print /x $REG	:	affiche le contenu du registre REG en hexadécimal
print /t $REG	:	affiche le contenu du registre REG en binaire
print /c $REG	:	affiche le contenu du registre REG sous forme de caractère
print /a $REG	:	affiche le contenu du registre REG sous forme d'adresse
printf "DESC",OBJ	:	affichage à la C
x /NFU ADDR	:	affichage du contenu de la mémoire à l'adresse ADDR N est le nombre d'unité à afficher F est le format d'affichage U indique le groupement : b pour 1 octet, h pour 2 octets et w pour 4 octets

Commandes d'aide

help	:	affiche l'aide
info program	:	affiche des informations sur le programme
info functions	:	affiche la liste des fonctions définies
info variables	:	affiche les variables et les symbols prédéfinies
info registers	:	affiche les informations sur les registres
info breakpoints	:	affiche les informations sur les points d'arrêts

Bibliographie

[1] Intel Corporation. *Intel Architecture Software Developer's Manual : Basis Architecture*, 1997.

[2] Intel Corporation. *Intel Architecture Software Developer's Manual : Instruction Set Reference*, 1997.